django-easymode Documentation

Release 0.6.1

Lars van de Kerkhof

April 05, 2012

CONTENTS

Easymode is an aspect oriented toolkit that helps making xml based flash websites easy. The tools included in the toolkit to help you make these kind of sites include:

CHAPTER

ONE

INTERNATIONALIZATION AND LOCALIZATION OF DJANGO MODELS

1.1 Internationalization of models

Django supports internationalization of text in templates and code by means of gettext. For internationalization of dynamic model data, easymode offers simple decorators to enable internationalized fields.

The only requuirement fields have to satisfy to be able to be internationalised by easymode, is that their `to_python()` method may not access `self`.

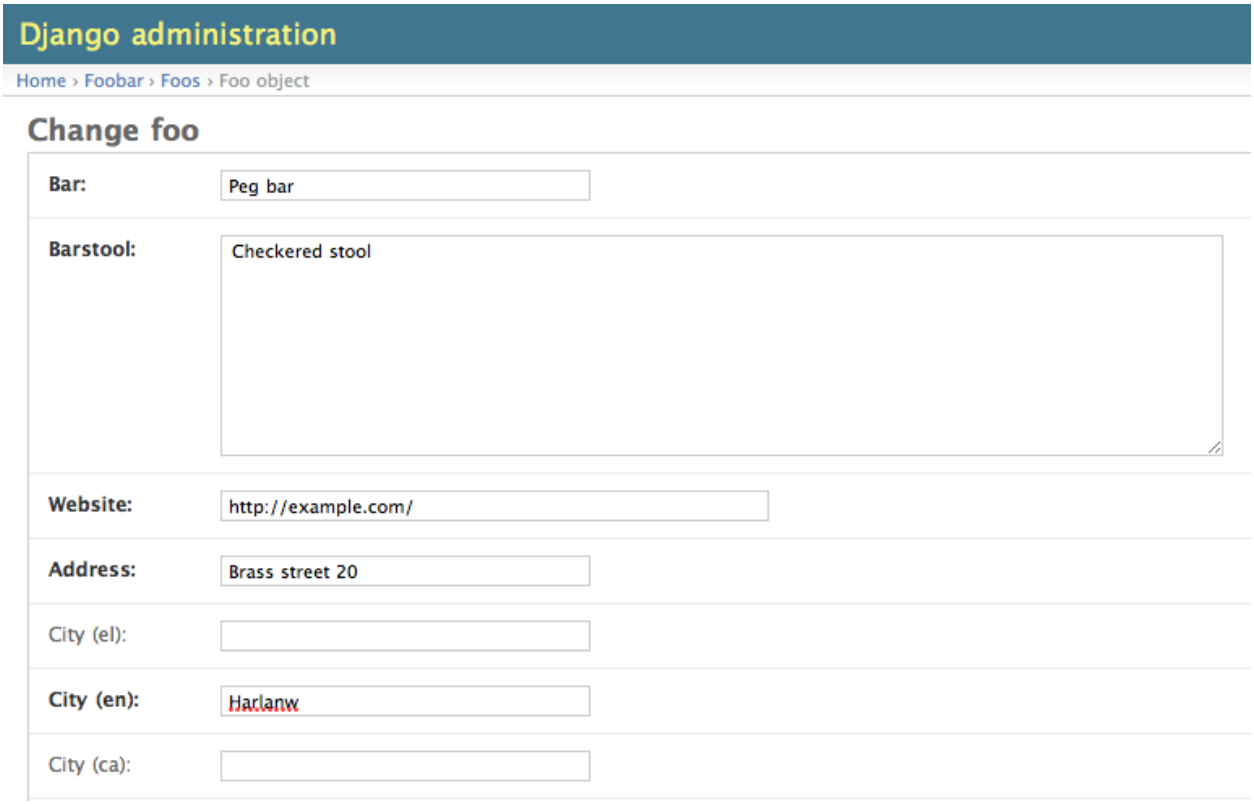
suppose we have the following model.

```
from django.db import models

class Foo(models.Model):
    bar = models.CharField(max_length=255, unique=True)
    barstool = models.TextField(max_length=4)
    website = models.URLField()
    address = models.CharField(max_length=32)
    city = models.CharField(max_length=40)
```

In different languages the city could have a different name, so we would like to make it translatable (eg. internationalize the city field). This can be done using the `I18n` decorator. Decorating the model as follows makes the field translatable:

```
from django.db import models
from easymode.i18n.decorators import I18n

@I18n('city')
class Foo(models.Model):
    bar = models.CharField(max_length=255, unique=True)
    barstool = models.TextField(max_length=4)
    website = models.URLField()
    address = models.CharField(max_length=32)
    city = models.CharField(max_length=40)
```

Now the `city` field is made translatable. As soon as you register this model with the admin, you will notice this fact. Depending on how many languages you got in `LANGUAGES` this is how your change view will look:



While useful, the interface can become very cluttered when more fields need to be internationalized. To make the interface less cluttered the admin class that belongs to the model, can be *Localized* making it show only the fields in the current language.

1.2 Localization of models in django admin

As there are several options to register a model for inclusion in django's admin, there are also several options to localize the admin classes.

The simplest way to make a model editable in the admin is:

```
from django.contrib import admin
from foobar.models import Foo

admin.site.register(Foo)
```

Since the admin class is implicit here, there is no way we can localize the admin class this way. The next simplest way is:

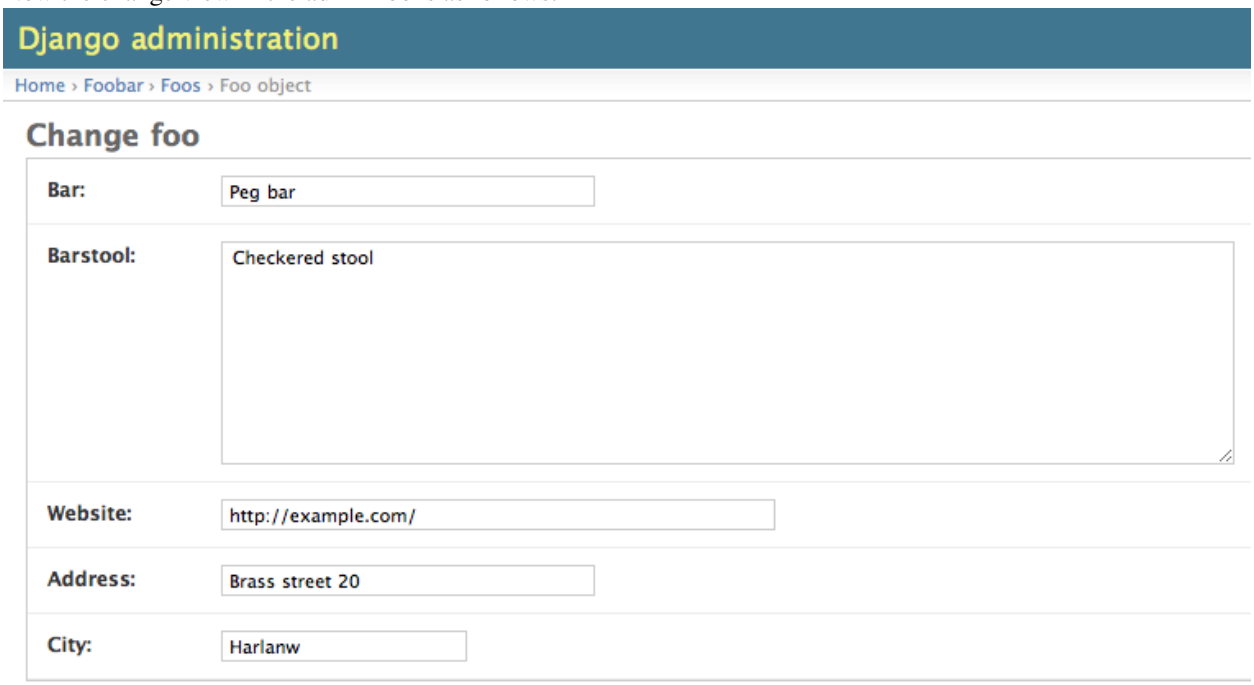
```
from django.contrib import admin
from foobar.models import Foo

admin.site.register(Foo, models.ModelAdmin)
```

Here the admin class is explicit, so we can modify it. The way this is done is by using the `L10n` class decorator:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n
from foobar.models import Foo
```

```
admin.site.register(Foo, L10n(Foo, models.ModelAdmin))
```

Note that the decorator needs the model to determine which fields are localized, so it must be passed as a parameter. Now the change view in the admin looks as follows:



All the 'city' fields are hidden, except for the field in the current language. Note That all fields which can be translated are marked with . To edit the content for the other languages, the current language must be switched. Please refer to *Translation of database content* for more details.

There is one more way a models can be registered for the admin and that is by creating a new descendant of `ModelAdmin` for a specific model. You can now also use the `L10n` decorator with the new class decorator syntax:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n

from foobar.models import Foo

@L10n(Foo)
class FooAdmin(admin.ModelAdmin):
    """Generic Admin class not specific to any model"""
    pass

admin.site.register(Foo, FooAdmin)
```

Note that you still have to pass the model class as a parameter to the decorator.

For admin classes that specify the `model` attribute you can leave that out:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n

from foobar.models import Foo

@L10n
```

```
class FooAdmin(admin.ModelAdmin):
    """Admin class for the Foo model"""
    model = Foo

admin.site.register(Foo, FooAdmin)
```

As you can see there isn't much to making models translatable this way.

1.3 Inline and GenericInline ModelAdmin

All easymode's localization mechanisms fully support django's flavors of `InlineModelAdmin`, both normal and generic. While there is no need to register these types of ModelAdmin classes, you still need to decorate them with `L10n` if you need them to be localized.

1.4 Fieldsets are also supported

`fieldsets` are supported for admin classes decorated with `L10n`. However `fields` is not supported, because easymode uses it to hide fields. Since you can do the exact same thing with fieldsets, this should not be a problem.

CHAPTER

TWO

TRANSLATION OF DATABASE CONTENT

When using the *i18n* and *l10n* features of easymode, you can use gettext's standard translation features to translate all the database content.

2.1 Automatic catalog management

If the *MASTER_SITE* directive is set to True, every time a model decorated with `I18n` is saved, easymode will add an entry to the corresponding gettext catalog. (for all the options related to the location of the catalogs please refer to *Easymode settings*). Additional control on what models should auto update the catalog is offered by *AUTO_CATALOG*.

For each language in your `LANGUAGES` directive, a catalog will be created. This way you can translate all the content using something like poedit or rosetta. This is especially convenient when a new site is created, for the first *big batch* of translations.

For modifications afterward, you can just use the admin interface, which will show the translations from the gettext catalog if they exist.

2.2 TAKE CARE

The translation mechanism using gettext is best used when a site is initially going to be translated to other languages. After this fase, content will most likely be edited directly in the admin interface, and you will encounter the issues described in *Database is bigger than gettext*. It takes proper planning to make full use of the gettext capabilities of easymode.

In effect any changes made to the gettext catalog after editors are changing content in the admin interface has a very low probability of being shown on the website. [1]

The proper workflow is:

- edit and add base content of the website, *ALL OF IT* and make sure you don't want to modify it anymore.
- translate content using gettext, and *COMPLETELY STOP ALL EDITING, JUST LOCK UP THE SITE DURING TRANSLATION!!!!!* (because of *Database is bigger than gettext*)

[1] Obviously, other gettext catalogs, generated from static content, that are not managed by easymode are unaffected.

- edit and modify all you like in the admin, all translations will be there. [2]

If you choose to deviate from this workflow be sure to understand all the next topics and learn how to use *easy_reset_language*.

2.3 Translation mechanism explained

It is important to realise, that allthough you can make translations using gettext, the catalog is not the only place translations are stored. The *I18n decorator* not only registers a model for catalog management, it also modifies the model.

suppose we have a model as follows:

```
@I18n('bar')
class Foo(models.Model):
    bar = models.CharField(max_length=255)
    foobar = models.TextField()
```

Normally the database would look like this:

```
CREATE TABLE "foobar_foo" (
    "id" integer NOT NULL PRIMARY KEY,
    "bar" varchar(255) NOT NULL,
    "foobar" text NOT NULL
)
```

The *I18n decorator* modiefies the model, given we've got both 'en' and 'yx' in out `LANGUAGES` directive this is what the model would look like on the database end:

```
CREATE TABLE "foobar_foo" (
    "id" integer NOT NULL PRIMARY KEY,
    "bar_en" varchar(255) NULL,
    "bar_yx" varchar(255) NULL,
    "foobar" text NOT NULL
)
```

On the model end you would not see this, because you will still access `bar` like this:

```
>>> m = Foo.objects.get(pk=1)
>>> m.bar = 'hello'
>>> print m.bar
hello
```

Any field that is internationalized using the *I18n decorator* will always return the field in the current languge, both on read and on write.

2.4 Database is bigger than gettext

Only when a field is empty (`None`) in the database for the current language, the gettext catalog will be consulted for a translation

[2] Watch out when you completely replace existing content in the *MSGID_LANGUAGE*. The *MSGID_LANGUAGE* is used for the message id's in the catalogs. When you completely replace the existing message id with something different, gettext will see that as adding a new message instead of changing an existing message. When this happens, translations can nolonger be associated with the new message and all languages will fall back to the new message id. Unless the content is allready saved in the database (*Database is bigger than gettext*).

This way, a model has exactly the same semantics as before, in that we can read and write to the property, the way we defined it in it's declaration. We still get the gettext goodies, which is nice when large ammounts of text must be translated.

If the gettext catalog would be the only place where the translations would be stored, having proper write semantics would become very difficult.

Example:

```
>>> from django.utils.translation import activate

>>> m = Foo()
>>> m.bar = 'hello'
>>> m.bar
'hello'
>>> activate('yx')
>>> m.bar
'hello'
>>> m.bar = 'xy says hello'
>>> m.bar
'xy says hello'
>>> activate('en')
>>> m.bar
'hello'
```

What you'll notice is that `m.bar` is allready available in the language 'yx' even though we did't specify it's value yet. This is because the normal behaviour of gettext is to return the `msgid` if the `msgstr` is not yet available. This is because the value for `m.bar` in langugae 'yx' was resolved as follows:

- see if the database value bar_yx is not null, if so return bar_yx
- see if the `msgstr` for 'hello' (The value of `m.bar` in the *MSGID_LANGUAGE*) exists if so return uget-text('hello')
- otherwise return the value in the *fallback language*

2.5 Importing translations is implicit

One thing that follows from the mechanics as described above, is that there is no need to explicitly import translations from gettext catalogs into the database.

Importing does take place however, each time a model is saved in the admin, the translations are written to the database.

This is because the translations from the gettext catalog *ARE* displayed in the admin, which means they *ARE* present in the form, but since the database column itself is *EMPTY* it will be marked as a change and written to the appropriate field.

This implicit import could pose a problem. If for example a model was edited in the admin, *BEFORE* the gettext catalog was properly translated and imported, it could be that the wrong value, from some *fallback language* got written to the database. Because the database get's precedence over the gettext catalog, the new translation would never show up.

This inconvenience can be resolved using the *easy_reset_language* command

CHAPTER

THREE

XSLT IS FOR FLASH

Most of the data being transferred to a flash frontend is in xml. This is both because xml is very well supported by Flash (e4x) and because hierarchical data is easily mapped to xml. Most data used for flash sited is hierarchical in nature, because the display list -flash it's version of html's DOM- is hierarchical as well.

What easymode tries to do is give you a basic hierarchical xml document that mirrors your database model, which you can then transform using xslt [1].

3.1 Why Xslt?

Xslt is a functional programming language, specifically designed to be used to transform one type of xml into another. So if we can reduce django's template rendering process to transforming one type of xml to another, xslt would be a dead on match for the job.

In fact we can. Easymode comes with a couple of serializers. These serializers differ from the normal django serializers, in that they treat a foreign key relation as a child parent relation. So while django's standard serializers output is flat xml, easymode's serializers output hierarchical xml.

3.2 Relations must be organized as a DAG

In order for easymode to be able to do it's work, the model tree should be organised as a DAG. If you have any cyclic relations, the serializer will get into an infinite loop and crash.

Most of the time you don't really need the cyclic relation at all. You just need to do some preprocessing of the data. You can render a piece of xml yourself, without using easymode's serializers and pass it to the xslt, see *Injecting extra data into the XSLT*.

3.3 Getting xml from a model

There are several ways to obtain such a hierarchical xml tree from a django model. The first is by decorating a model with the `toxml()` decorator:

```
from easymode.tree.decorators import toxml

@toxml
class Foo(models.ModelAdmin):
```

[1] Xslt requires a python xslt package. Easymode can work with lxml , libxslt and libxsltmod

```
    title = models.CharField(max_length=255)
    content = TextField()

class Bar(models.ModelAdmin):
    foo = models.ForeignKey(Foo, related_name=bars)

    label = models.CharField(233)
```

The `Foo` model has now gained a `__xml__` method on both itself as on the queryset it produces. Calling it will produce hierarchical xml, where all inverse relations are followed. (Except for `ManyToManyField`, they are not supported).

The preferred method for calling the `__xml__` method is by it's function:

```
from easymode.tree import xml

foos = Foo.objects.all()
rawxml = xml(foos)
```

3.4 Getting xml from several queries

The next option, which can also be used with multiple queries, is use the `XmlQuerySetChain`:

```
from easymode.tree import xml
from easymode.tree.query import XmlQuerySetChain

foos = Foo.objects.all()
qsc = XmlQuerySetChain(foos)
rawxml = xml(qsc)
```

Normally you would use the `XmlQuerySetChain` to group some `QuerySet` objects together into a single xml:

```
from easymode.tree import xml
from easymode.tree.query import XmlQuerySetChain

foos = Foo.objects.all()
hads = Had.objects.all()

qsc = XmlQuerySetChain(foos, hads)
rawxml = xml(qsc)
```

3.5 Using xslt to transform the xml tree

Now you know how to get the xml as a tree from the models, it is time to show how xslt can be used to transform this tree into something a flash developer can use for his application.

Easymode comes with one xslt that can give good results, depending on your needs:

```
from easymode.xslt.response import render_to_response

foos = foobar_models.Foo.objects.all()
return render_to_response('xslt/model-to-xml.xsl', foos)
```

The `render_to_response()` helper function will take an xslt as a template and a `XmlQuerySetChain` or a model/queryset decorated with `toxml()` to produce it's output. Additionally you can pass it a `dict` containing

xslt parameters. You have to make sure to use `prepare_string_param()` on any xslt parameter that should be passed to the xslt processor as a string.

Other helpers can be found in the `easymode.xslt.response` module.

CHAPTER

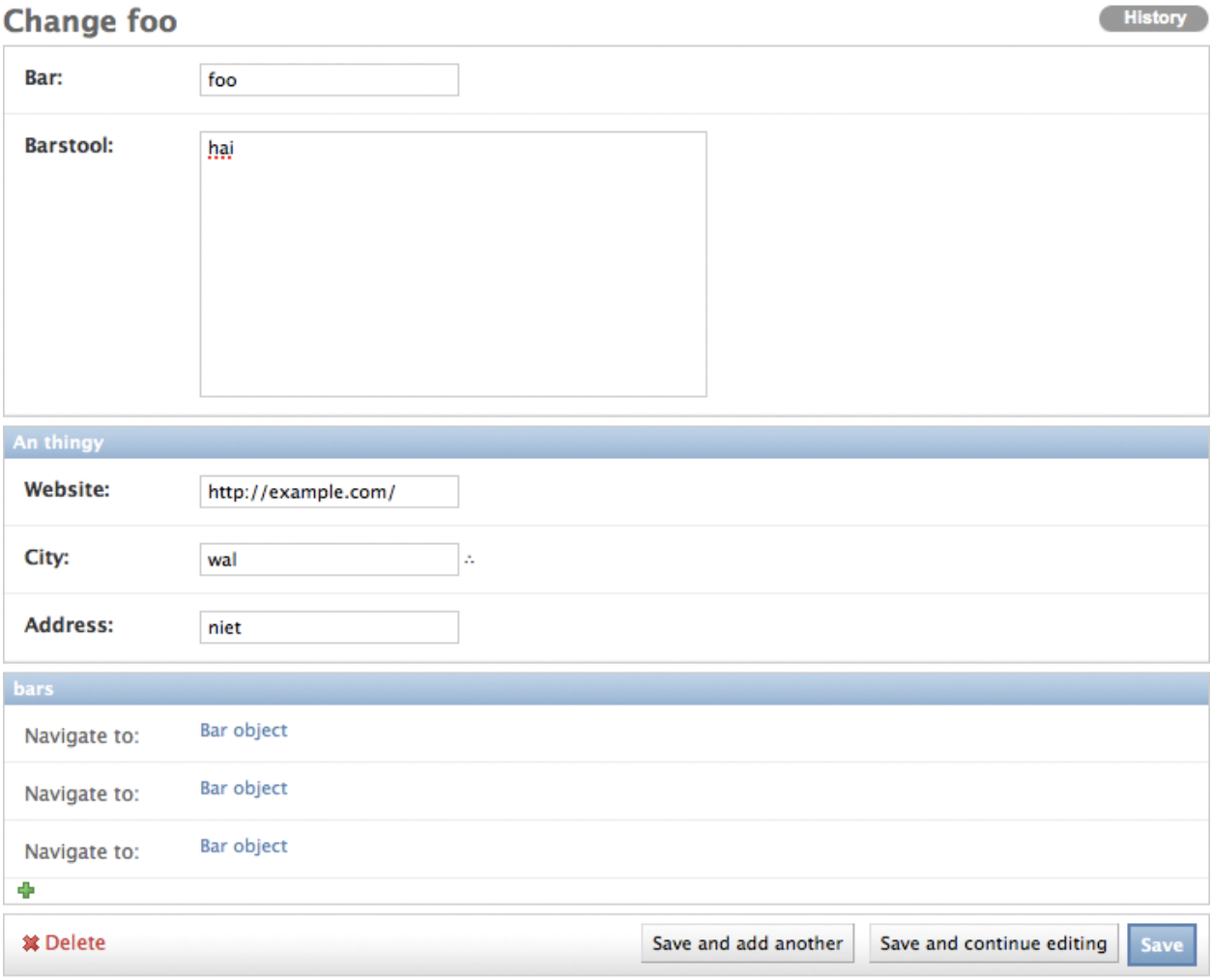
FOUR

ADMIN SUPPORT FOR MODEL TREES WITH MORE THAN 2 LEVELS OF RELATED ITEMS

Easymode has full admin support. Since content easymode was designed to handle is heavy hierarchic, easymode can also support this in the admin.

The single most annoying problem you will encounter when building django apps, is that after you discovered the niceties of `inlines`, you find out that only 1 level of `inlines` is supported. It does not support any form of recursion.

Easymode can not make InlineModelAdmin recursive either, because that would become a mess. What is *can* do, is display links to all related models. This way you have them in reach where you need them. There is no need to go back to the admin and select a different section to edit the related models.

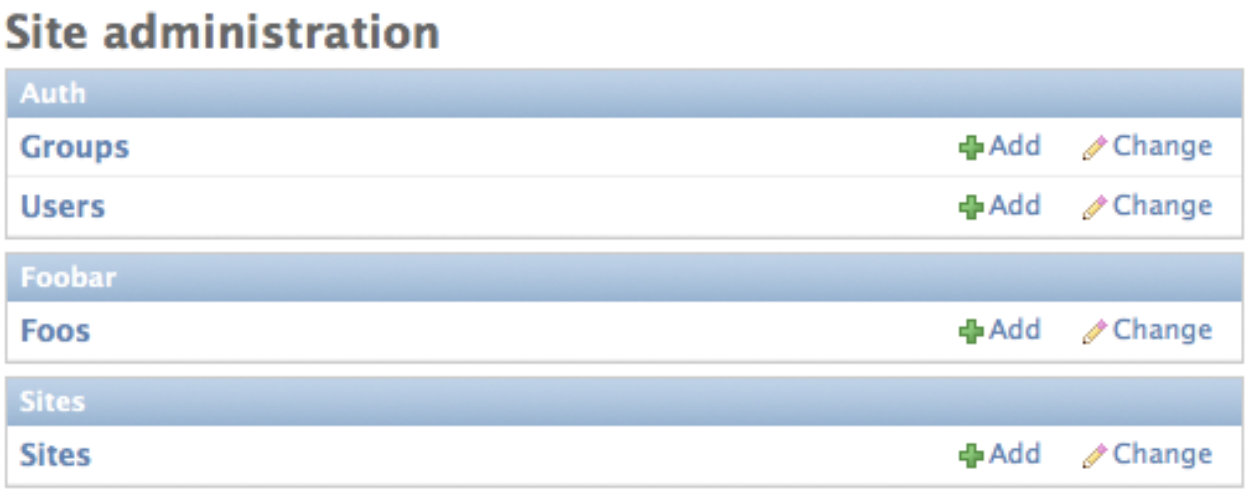


In the above picture, at the bottom of the `Bars` fieldset, there is a small + button. Using this button you can create new `Bar` objects which have a relation to the current `Foo` object. Just like with foreign key fields, the + button opens a popup in which you can create a new related item.

The items above the + button are all `Bar` objects that have a foreign key which points to the current `Foo` object. Clicking them will let you edit them.

4.1 Implementing the tree

To implement the tree first of all, you have to ensure that `easymode` comes before `django.contrib.admin` in the `INSTALLED_APPS` section of your settings file. This is because easymode needs to override the *admin/index.html* template. Since the related items that point to `Foo` can now be accessed from the `foo` change_view, it is nolonger needed that `Bar` is displayed in editable models list of the `Foobar` app. Just like InlineModelAdmin we want the 'inlined' models to be excluded from the app list.



This is how we want the `Foobar` app listing to look, with `Foo` visible and `Bar` excluded from the listing. In fact, that is what you can do with the `ModelAdmin` classes inside `easymode.tree.admin.relation`, as long as you make sure that the *admin/index.html* template is read from the `easymode` templates folder.

This is how the admin is defined to get the screenshots:

```
from django.contrib import admin
from easymode.i18n.admin.decorators import L10n
from easymode.tree.admin.relation import *

from foobar.models import Foo, Bar

@L10n
class FooAdmin(ForeignKeyAwareModelAdmin):
    """Admin class for the Foo model"""
    model = Foo
    invisible_in_admin = False

    fieldsets = (
        (None, {
            'fields': ('bar', 'barstool')
        }),
        ('An thingy', {
            'fields': ('website', 'city', 'address')
        }),
    )

class BarAdmin(InvisibleModelAdmin):
    model = Bar
    parent_link = 'foo'

admin.site.register(Foo, FooAdmin)
admin.site.register(Bar, BarAdmin)
```

As you can see the `ModelAdmin` classes used are `InvisibleModelAdmin` and `ForeignKeyAwareModelAdmin`.

`ForeignKeyAwareModelAdmin` is aware of the models that have a `ForeignKey` pointing to the model which it makes editable.

In this case, `FooAdmin` makes `Foo` editable, and `Bar` has a `ForeignKey` which points to `Foo`. `FooAdmin` is fully aware of this! In fact it will make you aware as well, because it will display all the related `Bar` models in `Foo`'s

`change_view()`.

As said we'd like to have `Bar` be invisible in the `Foobar` app listing. That is where `InvisibleModelAdmin` comes into play. Using `InvisibleModelAdmin` instead of a normal `ModelAdmin` will hide the model from the app listing.

You could even use a `ForeignKeyAwareModelAdmin` in place of the `InvisibleModelAdmin` because it can be made invisible as well. Using these 2 `ModelAdmin` classes, mixed with regular InlineModelAdmin you can create deep trees and manage them too.

CHAPTER

FIVE

BASIC APPROVAL SUPPORT FOR MODELS

Easymode comes with `easypublisher`, a very simple approval application. It uses django-reversion to store drafted content. This has the very nice side effect that all drafts are in your history.

There is only one layer of approval, either you've got publishing rights or you don't. Anyone with publisher rights can move content from draft to published, as long as they've got permission to modify the content.

To use the publisher you have to include `easymode.easypublisher` in your `INSTALLED_APPS`. After that, you may use `EasyPublisher` instead of `ModelAdmin` as follows:

```
from django.contrib import admin
from foobar.models import *
from easymode.easypublisher.admin import EasyPublisher

class FooAdmin(EasyPublisher):
    model = Foo

admin.site.register(Foo, FooAdmin)
```

A new permission will be added `easypublisher.can_approve_for_publication` if some body does *NOT* have this permission, all their changes will only be saved as versions and never in the database. All people who *DO* have this permission can view the list of *drafts*, load them and save them, which means they are published. All your drafts and versions will be kept track of by django-reversion.

In case you want to use easypublisher together with `easymode.tree.admin.relation` you will find that multiple inheritance doesn't work due to conflicts. Instead, use `easymode.easypublisher.admin.EasyPublisherFKAModelAdmin` where you would use `ForeignKeyAwareModelAdmin` and `easymode.easypublisher.admin.EasyPublisherInvisibleModelAdmin` where you would use `InvisibleModelAdmin`.

More info about these admin classes is in *Admin support for model trees with more than 2 levels of related items*.

5.1 Easypublisher templatetag `draft_list_items()`

`draft_list_items()` is a templatetag that can be used to show all drafts that need approval as a list of links to these drafts. You could include it in your admin template somewhere.

use like this:

```
{% load 'easypublisher' %}

<ul>
{% draft_list_items %}
</ul>
```

This will render as a list of links to all unapproved drafts.

The best way to learn how easymode works, is to read the above topics in sequence and then look at the *Example*.

CHAPTER

SIX

VERSION NAMING CONVENTION

- Each update to the development status will increase the first digit. (eg beta or alpha or production ready)
- Each new feature will increase the second digit.
- Each bugfix or refactor will increase the last digit

CHAPTER

SEVEN

EXAMPLE

Easymode comes with an example app which is available from github:

http://github.com/LUKKIEN/django-easymode/

To run the example app, you must clone the repository, install the dependencies and initialize the database:

```
git clone http://github.com/LUKKIEN/django-easymode.git
cd django-easymode
pip install -r requirements.txt
cd example
python manage.py syncdb
python manage.py loaddata example_data.xml
python manage.py runserver
open http://127.0.0.1:8000/
```

CHAPTER

EIGHT

UNSUPPORTED DJANGO FEATURES

The following features, which django supports, are not supported by easymode:

- `unique_together`
- `unique_for_date`, `unique_for_month`, `unique_for_year`
- `django.contrib.admin.ModelAdmin.fields`, use `django.contrib.admin.ModelAdmin.fieldsets` instead.
- Automatic serialization of `ManyToManyField`. The model tree should be a DAG.
- Inheritance for models is restricted to `abstract` base classes. This is a direct result of the fact that `OneToOneField` are *not* supported by the serializer.

All these features are not supported because the ammount of work to have them was greater than the benefit of having them.

CHAPTER

NINE

ADDITIONAL SUBJECTS

9.1 Easymode settings

9.1.1 AUTO_CATALOG

Easymode can manage a gettext catalog with your database content for you. If `AUTO_CATALOG` is `True`, easymode will add every new object of a model decorated with `I18n` to the gettext catalog.

When existing content is updated in the *MSGID_LANGUAGE* on the *MASTER_SITE*, gettext will try to updated the msgid's in all the languages. Therefor keeping the mapping between original and translation. There is a limit on the ammount of change, before gettext can nolonger identify a string as a change in an existing msgid. For example:

```
# in the english django.po
#: main.GalleryItem.title_text:32
msgid "I've got a car"
msgstr ""

# in the french django.po
#: main.GalleryItem.title_text:32
msgid "I've got a car"
msgstr "J'ai une voiture"

#now we update the main.GalleryItem.title_text in the db in english

# in the english django.po
#: main.GalleryItem.title_text:32
msgid "I've had a car"
msgstr ""

# gettext will now also update the message id in french so the link
between original and translation keeps existing.

# in the french django.po
#: main.GalleryItem.title_text:32
msgid "I've had a car"
msgstr "J'ai une voiture"
```

The location of the catalog canbe controlled using *LOCALE_DIR* and *LOCALE_POSTFIX*,

example:

```
AUTO_CATALOG = False
```

With the above settings, no catalogs are managed automatically by easymode. You have to manually generate them using *easy_locale*.

AUTO_CATALOG can also be used when you only need *some* (but not all) of the internationalised models to auto update the catalog. For this to work you need to set *AUTO_CATALOG* to `False` in settings.py:

```
AUTO_CATALOG = False
```

Then somewhere else, for example in your admin.py or models.py you can turn on automatic catalog updates for specific models:

```
from models import News
import easymode.i18n

easymode.i18n.register(News)
```

Now only the `News` model will automatically update the catalog, but other models will leave it alone. See `easymode.i18n.register()` for more info.

Ofcourse, for this to work you must have *MASTER_SITE* set to `True`.

9.1.2 MASTER_SITE

The `MASTER_SITE` directive must be set to `True` if a gettext catalog should be automatically populated when new contents are created. This way all contents can be translated using gettext. You can also populate the catalogs manually using the *easy_locale* command.

In a multiple site context, you might not want to have all sites updating the catalog. Because the content created on some of these sites might not need to be translated because it is not used on any other sites. Content can flow from 'master site' to 'slave site' but not from 'slave site' to 'slave site'.

example:

```
MASTER_SITE = True
```

9.1.3 MSGID_LANGUAGE

The `MSGID_LANGUAGE` is the language used for the message id's in the gettext catalogs. Only when a content was created in this language, it will be added to the gettext catalog. If `MSGID_LANGUAGE` is not defined, the `LANGUAGE_CODE` will be used instead. The msgid's in the gettext catalogs should be the same for all languages.

This setting should be used when there are different sites, each with a different `LANGUAGE_CODE` set. These sites can all share the same catalogs.

example:

```
MSGID_LANGUAGE = 'en'
```

9.1.4 FALLBACK_LANGUAGES

The `FALLBACK_LANGUAGES` is a dictionary of values that looks like this:

```
FALLBACK_LANGUAGES = (
    'en': [],
    'hu': ['en'],
    'be': ['en'],
```

```
    'ff' :['hu','en']
)
```

Any string that is not translated in 'ff' will be taken from the 'hu' language. If the 'hu' also has no translation, finally it will be taken from 'en'.

9.1.5 LOCALE_DIR

Use the `LOCALE_DIR` setting if you want all contents to be collected in a single gettext catalog. If `LOCALE_DIR` is not specified, the contents will be grouped by app. When a model belongs to the 'foo' app, new contents will be added to the catalog located in `foo/locale`.

You might not want to have the dynamic contents wriiten to your app's locale, if you also have static translations. You can separate the dynamic and static content by specifying the *LOCALE_POSTFIX*.

example:

```
PROJECT_DIR = os.dirname(__file__)
LOCALE_DIR = os.path.join(PROJECT_DIR, 'db_content')
LOCALE_PATHS = (join(LOCALE_DIR, 'locale'), )
```

(Note that by using `LOCALE_PATHS` the extra catalogs are loaded by django).

9.1.6 LOCALE_POSTFIX

The `LOCALE_POSTFIX` must be used like this:

```
LOCALE_POSTFIX = '_content'
```

Contents that belong to models defined in the 'foo' app, will be added to the catalog located at `foo_content/locale` instead of `foo/locale`.

9.1.7 USE_SHORT_LANGUAGE_CODES

Easymode has some utilities that help in having sites with multiple languages. `LocaliseUrlsMiddleware` and `LocaleFromUrlMiddleWare` help with adding and extracting the current language in the url eg:

http://example.com/**en**/page/1

When having many similar languages in a multi site context, you will have to use 5 letter language codes:

en-us en-gb

These language codes do not look pretty in an url:

http://example.com/en-us/page/1

and they might even be redundant because the country code is allready in the domain extension:

http://example.co.uk/en-gb/page/1

When `USE_SHORT_LANGUAGE_CODES` is set to `True`, the country codes are removed in urls, leaving only the language code. This means the url would say:

http://example.com/en/page/1

even when the current language would be 'en-us'.

THIS DIRECTIVE ONLY WORKS WHEN THERE IS NO AMBIGUITY IN YOUR `LANGUAGES` DIRECTIVE.

This means i can not have the same language defined twice in my `LANGUAGES`:

```
LANGUAGES = (
    'en-us' : _('American English'),
    'en-gb' : _('British'),
)
```

This will **NOT** work because both languages will be displayed in the url as 'en' which is ambiguous.

9.1.8 SKIPPED_TESTS

It might be that some tests fail because you've got some modules disabled or you can not comply to the test requirements. This is very annoying in a continuous integration environment. If you are sure that the failing tests cause no harm to your application, they can be disabled.

`SKIPPED_TESTS` is a sequence of test case names eg:

```
SKIPPED_TESTS = ('test_this_method_will_fail', 'test_this_boy_has_green_hair')
```

will make sure these 2 tests will not be executed when running the test suite.

9.2 Management Commands

9.2.1 easy_locale

Easy locale will update the gettext catalogs with content from the database. This can be specific to a single app or model.

Help output:

```
Usage: manage.py easy_locale [options]


        easy_locale <targetdir> <applabel>

        Will create a folder locale in targetdir with locales parsed
        from the models in applabel.

        example:
        ./manage.py easy_locale myapp myapp

        will create myapp/locale/ with po files in it.
```

9.2.2 easy_reset_language

This command will clear the database fields in one langugage for a specific app or model, so the translation will once again come from the catalog, instead of the database.

Help output:

```
Usage: manage.py easy_reset_language [options]


        easy_reset_language <target locale> <app>

        will clear all fields for the locale in question so the values will be read from
        the locale again.

        example:
        ./manage.py easy_reset_language en myapp.mymodel

        This will clear myapp.mymodel in the en locale so all values
        will be fetched by gettext instead of being overridden.
```

9.3 Easyfilters

Easymode comes with 3 templatetags that can be used to modify existing templates so they can be used in a multilingual environment.

9.3.1 `strip_locale()`

`strip_locale()` will have an url as an argument and if there is a locale in the url, it will be stripped:

```
{% load 'easyfilters' %}

{{ 'http://example.com/en/greetings'|strip_locale }}
```

this will render as: `http://example.com/greetings` so the 'en' part will be removed from the url.

You can use this filter in combination with `LocaliseUrlsMiddleware`. The middleware will add the current language to any urls that do not have the language code in the url yet.

9.3.2 `fix_locale_from_request()`

Fixes the language code as follows:

If there is only one language used in the site, it strips the language code. If there are more languages used, it will make sure that the url has the current language as a prefix.

usage:

```
{% load 'easyfilters' %}

{{ 'http://example.com/en/greetings'|fix_locale_from_request:request.LANGUAGE_CODE }}
```

Suppose `request.LANGUAGE_CODE` was 'ru' then the output would become:

```
http://example.com/ru/greetings
```

Suppose `settings.LANGUAGES` contained only one language, the output would become:

```
http://example.com/greetings
```

You probably do not need this templatetag if you are using `LocaliseUrlsMiddleware`.

9.3.3 fix_shorthand()

Use this if you want to use *USE_SHORT_LANGUAGE_CODES*.

`fix_shorthand()` will always return the correct locale to use in an url, depending on your settings of *USE_SHORT_LANGUAGE_CODES*.

usage:

```
{% load 'easyfilters' %}

{{ request.lANGUAGE_CODE|fix_shorthand }}
```

Suppose `request.LANGUAGE_CODE` is 'fr-be' and *USE_SHORT_LANGUAGE_CODES* is set to `True`, the output would become:

```
fr
```

If *USE_SHORT_LANGUAGE_CODES* is set to `False` the output would be:

```
fr-be
```

If `request.LANGUAGE_CODE` is not a five letter language code, nothing happens.

9.4 easymode.middleware

9.4.1 Google Analytics

Easymode has middleware to support caching in combination with google analytics. Google analytics updates a session cookie on each request. Because django's `SessionMiddleware` places cookie in it's vary header, *you will save every single request to the cache* if you use it.

9.4.2 Internationalization related middleware

9.5 Injecting extra data into the XSLT

If you want to have some extra data passed to the xslt, which can not be obtained by the serializer you can make some view helpers that create xml and pass it as a stringparam to the xslt.

Reasons why you would need this:

- You've got a model that has a foreign key to itself. You need this if you want some kind of hierarchical page tree or something. You might want to put the self referencing `ForeignKey` to `serialize=False`. This way it can not mess up the serializer, but you don't have a hierarchic structure in your xml.
- You pull data from an external source.
- You have to do some processing on the models before they get turned into xml.
- You have some data not coming from models that needs to be passed to the xslt.

In all these cases you can use `XmlPrinter` to make some well formed unicode safe xml you can feed to the xslt.

Here is an example where some static strings get passed to the xslt. These strings are translatable using django's regular i18n mechanism, but they are not in the database:

```
from django.utils.translation import ugettext as _

stringlib = dict(
    close_button = _('Close'),
    next_button = _('Next'),
    the_end = _("That's all folks")
)

def render_stringlib_xml():
    """Renders the stringlib xml"""
    stream = StringIO()
    xml = XmlPrinter(stream, settings.DEFAULT_CHARSET)
    xml.startElement('stringlib', {'id':'stringlib'})
    for (key, value) in stringlib.iteritems():
        xml.startElement(key, {})
        xml.characters(value)
        xml.endElement(key)
    xml.endElement('stringlib')

    byte_string = stream.getvalue()
    return byte_string.decode('utf-8')
```

Before you pass the rendered xml string, you should prepare it using `prepare_string_param()`:

```
from easymode.xslt import prepare_string_param as q
from easymode.xslt.response import render_to_response

params = {
    'stringlib' : q(render_stringlib_xml()),
}

qs = Foo.objects.all()

return render_to_response('xslt/model-to-xml.xsl', qs, params)
```

9.6 Release Notes

9.6.1 v0.6.1

- `DiocoreHTMLField` will now also show a tinymce editor when it is not internationalized.
- When there is a problem with monkey patching `django.db.models.SubfieldBase` easymode will throw an exception. (Monkey patch fixes http://code.djangoproject.com/ticket/12568).
- New field aded, `CSSField`, which allows specification of css classes for a rich text field, the css classes will appear in the xml as:

  ```
  style="class1,class2"
  ```

9.6.2 v0.6.0

- Django 1.2 is required for easymode as of v0.6.0.

- `get_real_fieldname()` now returns a string instead of `unicode`. This way a `dict` can be constructed using it's results as keys, and the dict can be turned into keyword arguments of `filter` when doing a query in a specific language.
- Small improvements in error handling when *AUTO_CATALOG* is `True`

9.6.3 v0.5.7

- Added `easymode.admin.fields.SafeTextField`, a textfield which strips all cariage returns before saving, which is required when using *Automatic catalog management*.
- Updated django requirement to v1.1.2 because python 2.6.5 will otherwise make the unit tests fail.

9.6.4 v0.5.6

- The example app now has a working fixture.

9.6.5 v0.5.5

- Special admin widgets are nolonger discarded by easymode (issue #3)

9.6.6 v0.5.4

- Some data files where not installed correctly by setup.py

9.6.7 v0.5.3

- Added *AUTO_CATALOG* setting, see *Automatic catalog management*.
- Fixed error in *easy_locale* when two properties in the same model have the same value (eg. title and subtitle are the same).

CHAPTER

TEN

GETTING EASYMODE

You can download easymode from:

http://github.com/LUKKIEN/django-easymode/downloads/

Or you can do:

- `pip install django-easymode`

Or: - `pip install -e git://github.com/LUKKIEN/django-easymode.git#egg=easymode`

Note the version number in the top left corner and use:

- `easy_install http://github.com/LUKKIEN/django-easymode/tarball/[VERSION]`

Which, if the version was v0.1.0 would become http://github.com/LUKKIEN/django-easymode/tarball/v0.1.0.

CHAPTER

ELEVEN

API DOCS